散1⃣️

git remote add test [git@github.com](mailto:git@github.com):test,

git remote add origin [git@github.com](mailto:git@github.com):za-zhusifen/note.git,

解释：

git remote add 的作用好像只是加个名称标签，如上面两句是加了origin标签（）和test标签（）

这些都记录在该本地git仓库的.git/config文件中：

～～～～～～～～～

Config

～～～～～～～～～

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

ignorecase = true

precomposeunicode = true

[remote "origin"]

url = git@github.com:za-zhusifen/note.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

[branch "master"]

remote = origin

merge = refs/heads/master

[remote "test"]

url = git@github.com:test

fetch = +refs/heads/\*:refs/remotes/test/\*

～～～～～～～～～

~~~~~~~~~~~~~~~

有些文件不需要放入Git中：

在项目的根目录添加一个名为.gitignore文件，内容如下:

# maven #

target/

# IDEA #

.idea/

\*.imi

# Eclipse #

.settings/

.metadata/

.project

.classpath

Servers/

这样，这些文件和文件夹就不用加进git管理里了。

散2:

A分支新建了一个文件afile，然后A merge到master，从master上merge到B分支，然后B分支上把afile文件删了，之后B分支merge到master，这时候master上没有了afile（这属于最新内容），当A分支上的afile没有新的变化（属于旧的内容），A分支merge到master或者反过来，这时候afile都会没了。这时候要用compare with 分支对比分支不同手动添加文件。

慕课网之git入门：

1.初始化本地版本库

进入一个文件夹

— git init

此时生成一个.git隐藏文件夹，初始化好本地版本库

2.添加文件到本地版本库

往文件夹中添加新的文件XX，文件XX在<未暂存区>/<工作区>，一个未跟踪文件

Changes not staged for commit:

modified: 文件XX

— git add [文件XX]

文件XX(有不同的文件)到了<暂存区>

Changes to be committed:

modified: 文件XX

— git commit -m “本次提交的描述”

文件XX到了<本地版本库>，默认到master分支？

3.查看仓库状态

— git status

4**.**工作流

— git log : 展示每次提交到<本地版本库>的日志:

commit f3bac05c30bd8a0af74acec827643da523a5d8b7（**<commit>**提交号）

Author: 朱思奋 <zhusifen@192.168.4.103>

Date: Sat Apr 6 23:23:12 2019 +0800

rm file

commit d1dbeab57a15df093bb7d161e2c093046cf7f9b3（**<commit>**提交号）

Author: 朱思奋 <zhusifen@192.168.4.103>

Date: Sat Apr 6 23:21:55 2019 +0800

4 bla bla

— **git add** 文件： <未暂存区> —增或减的变化更新到—> <暂存区>

— **git commit -m “**本次提交的描述**”** ：<暂存区> —覆盖—> <本地版本库>，每次提交就有一个<commit>提交号

— **git checkout —-** 文件名 **...** ：

**git checkout** 文件名 **...** ：

**git checkout HEAD --** 文件名 **...** ：<暂存区> —覆盖—> <未暂存区>

— **git checkout <commit>**提交号：<本地版本库>指定<commit>号的版本 —覆盖—> <暂存区>和<未暂存区(工作区）>

You are in 'detached HEAD' state. 分离状态

checkout到某个commit号后，HEAD指针变成了detached状态。这是不影响原有任何分支的一个commit号，若做些更改后提交，会产生另一个新的commit号，这些都不在某个分支上，当用命令 git branch <new-branch-name> commit号，后才会把这条新代码线放进一个新创建的分支里。

—**git branch <new-branch-name> commit**号 ： 用某个commit号创建一个新分支，新分支的内容就是这个commit号的内容。

—**git checkout --datch** 分支名 ：切换到指定分支的分离状态，默认为该分支下的最后一次提交的commit号

—**git checkout -b <new\_branch> commit**号：用某commit号创建新分支并切换到新分支

— **git checkout <commit>**提交号 **—-** 文件名 **...** ：针对指定文件，<本地版本库>指定<commit>号的版本 —覆盖—> <暂存区>和<未暂存区(工作区）>

—**git checkout -B <branch>** ：同名分支无法创建时，使用-B参数，那么就可以强制创建新的分支，并会覆盖掉原来的同名分支

— **git checkout** 分支名：切换到某个分支

—-**git branch** 分支名：新建一个分支

—-**git checkout -b** 分支名：新建分支并切换到这个分支，<暂存区>和<未暂存区>的文件不变。

— **git reset**

**git reset** 文件名 **...** ：<本地版本库>最新版本 —覆盖—> <暂存区>

—**git reset <commit>**提交号：

**git reset —-mixed** 版本号：mixed为default配置。

1.<本地版本库>中指定的版本 —覆盖—> <暂存区>的所有; 2.HEAD指针指向指定的版本号，当前<本地版本库>的当前版本为指定的版本。

——**git reset <commit>**提交号 **—-** 文件名 **...** ： 1.<本地版本库>中指定的版本的指定文件 —覆盖—> <暂存区>的指定文件；2.但HEAD指针不变，当前<本地版本库>的当前版本号不变。

——**git reset —-hard** 版本号：加**—-hard**的意思是，<本地版本库>中指定的版本 —覆盖—> <暂存区>和<未暂存区>。好像不能加”—- 文件名”的写法。

——**git reset —-soft <commit>**提交号：加**—-soft**的意思是，告诉Git重置HEAD到另外一个commit，但也到此为止。就说<本地版本库>的当前版本变到指定版本号。

——**git reset HEAD~2**让分支向后回退了两个提交。分支末端的两个提交现在变成了悬挂提交。也就是说，下次 Git 执行垃圾回收的时候，这两个提交会被删除。换句话说，如果你想扔掉这两个提交，你可以这么做。

--------

有时候同一个分支，远程的和本地的都被修改的面目全非了，如果想要把本地的替换成远程的，用下面的命令

*git* *fetch* --*all*

*git* *reset* --*hard* *origin/master* *(这里master要修改为对应的分支名)*

*git* *pull*

--------

— **git rm** 文件: 删除<未暂存区>和<暂存区>的文件，

同手动删除<未暂存区>文件，然后git add 文件。

— **git reset —-hard** 版本号：最终仓库 暂存区

(只要<working tree>还存在这个<版本号>，如果<working tree>重新生长，那之前的枝上的<版本号>就没了，如果只是回到某个版本，还没长新枝，旧枝的整个版本号还是都在的。)

5.其他

“--" 好像是用来分隔命令与文件的

Use '--' to separate paths from revisions, like this:

'git <command> [<revision>...] -- [<file>…]'

HEAD 通常会指向当前所在分支的最新版本

git checkout HEAD 文件：工作区和暂存区都会滚到 当前

**Revert**

Revert 撤销一个提交的同时会创建一个新的提交。这是一个安全的方法，因为它不会重写提交历史。比如，下面的命令会找出倒数第二个提交，然后创建一个新的提交来撤销这些更改，然后把这个提交加入项目中。

git checkout hotfix

git revert HEAD~2

如下图所示：

相比 git reset，它不会改变现在的提交历史。因此，git revert 可以用在公共分支上，git reset 应该用在私有分支上。

你也可以把 git revert 当作撤销已经提交的更改，而 git reset HEAD 用来撤销没有提交的更改。

就像 git checkout 一样，git revert 也有可能会重写文件。所以，Git 会在你执行 revert 之前要求你提交或者缓存你工作目录中的更改

—————-

安装完成后，还需要最后一步设置，在命令行输入：

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

注意git config命令的—global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

—————-

图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

不幸的是，Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的，如果要真正使用版本控制系统，就要以纯文本方式编写文件。

因为文本是有编码的，比如中文有常用的GBK编码，日文有Shift\_JIS编码，如果没有历史遗留问题，强烈建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

—————-

$ git **commit** -m "wrote a readme file"

[master (root-**commit**) eaadf4e] wrote a readme file

1 file changed, 2 insertions(+)

**create** mode 100644 readme.txt

简单解释一下git commit命令，-m后面输入的是本次提交的说明。嫌麻烦不想输入-m “xxx"行不行？实在不想输入说明的童鞋请自行Google

git commit命令执行成功后会告诉你，1 file changed：1个文件被改动（我们新添加的readme.txt文件）；2 insertions：插入了两行内容（readme.txt有两行内容）

—————-

git diff 文件：是查看工作区<未暂存区>与<暂存区>的区别。

git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式

**Git diff**

用于比较两次修改的差异

**1.1 比较工作区与暂存区**

git diff 不加参数即默认比较工作区与暂存区

**1.2 比较暂存区与最新本地版本库（本地库中最近一次commit的内容）**

git diff --cached  [<path>...]

**1.3 比较工作区与最新本地版本库**

　　　　　　git diff HEAD [<path>...]  如果HEAD指向的是master分支，那么HEAD还可以换成master

**1.4 比较工作区与指定commit-id的差异**

　　　　　　git diff commit-id  [<path>...]

**1.5 比较暂存区与指定commit-id的差异**

　　　　　　git diff --cached [<commit-id>] [<path>...]

**1.6 比较两个commit-id之间的差异**

　　　　　　git diff [<commit-id>] [<commit-id>]

**1.7 使用git diff打补丁**

git diff > patch //patch的命名是随意的，不加其他参数时作用是当我们希望将我们本仓库工作区的修改拷贝一份到其他机器上使用，但是修改的文件比较多，拷贝量比较大，

　　　　　　此时我们可以将修改的代码做成补丁，之后在其他机器上**对应目录下**使用 git apply patch 将补丁打上即可

　　　　　　git diff --cached > patch //是将我们暂存区与版本库的差异做成补丁

　　　　　　  git diff --HEAD > patch //是将工作区与版本库的差异做成补丁

　　　　　　git diff Testfile > patch//将单个文件做成一个单独的补丁

　　　　拓展：git apply patch 应用补丁，应用补丁之前我们可以先检验一下补丁能否应用，git apply --check patch 如果没有任何输出，那么表示可以顺利接受这个补丁

　　　　　　　另外可以使用git apply --reject patch将能打的补丁先打上，有冲突的会生成.rej文件，此时可以找到这些文件进行手动打补丁

学习无他法，唯有持之以恒

—————-

首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交1094adb...（注意我的提交ID和你的肯定不一样），上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100

—————-

版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了

—————-

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD从指向变换下，然后顺便把工作区的文件更新了。

—————-

Git提供了一个命令git reflog用来记录你的每一次命令

—————-

我们创建Git版本库时，Git自动为我们创建了唯一一个master分支

—————-

git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到git checkout命令

—————-

git ls-files命令是用来查看暂存区中文件信息。

二、常用参数

|  |  |
| --- | --- |
| **参数** | **说明** |
| --cached | 显示暂存区中的文件，git ls-files命令默认的参数 |
| -c | --cached的简写 |
| --deleted | 显示删除的文件 |
| -d | -deleted的简写 |
| --modified | 显示修改过的文件 |
| -m | --modified参数的简写 |
| --other | 显示没有被git跟踪的文件 |
| -o | --other的简写 |
| --stage | 显示mode以及文件对应的Blob对象，进而我们可以获取暂存区中对应文件里面的内容。 |
| -s | --stage简写 |

二、实例

**1.**如何查看暂存区中有哪些文件？

可以直接使用如下命令：

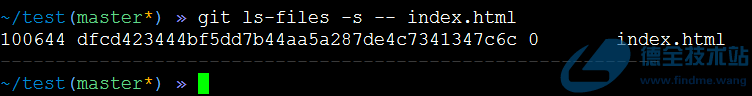
|  |  |
| --- | --- |
| 1 | git ls-files |



**1.**如何查看暂存区中**index.html**文件内容是什么？

首先，我们需要查查index.html文件对应的Blob对象，如下：

|  |  |
| --- | --- |
| 1 | git ls-files -s -- index.html |



然后通过Blob对象，查询index.html里面的内容：

|  |  |
| --- | --- |
| 1 | git cat-file -p dfcd42 |

—————-

.git/HEAD 写着HEAD指针的指向

.git/refs/heads/分支 写着分支的最后commit号

在切换分支的时候，HEAD也会相应的指向对应的分支引用

—————-

git checkout a

这时候，提示检出失败，git以为我们想检出仓库a。还记得为什么在第一步中，我们曾新建的文件a吗？这里终于派上用场了，由于仓库中还存在分支a，同时当前分支中又存在文件a，于是git傻傻分不清楚了。这时候怎么办？有两个办法，第一，我们在命名分支的时候要注意语义性，分支名要具有一定的意义，不能使用简单的a,b,c来命名，这样很容易导致分支名和文件名重复；第二，参照用法1，使用两个连字符来分隔。在目前的情形中，我们使用第二种方法吧。

git checkout —- a

—————-

checkout命令用法如下：

1. git checkout [-q] [<commit>] [--] <paths> ...

2. git checkout [<branch>]

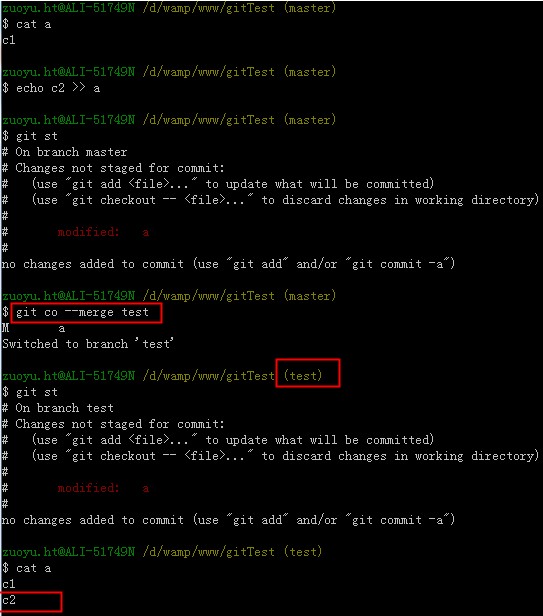
3. git checkout [-m] [ [-b | -- orphan ] <new\_branch>]  [start\_point]

—————-

https://www.cnblogs.com/kuyuecs/p/7111749.html

**5. git checkout --merge <branch>**

这个命令适用于在切换分支的时候，将当前分支修改的内容一起打包带走，同步到切换的分支下。



有两个需要注意的问题。

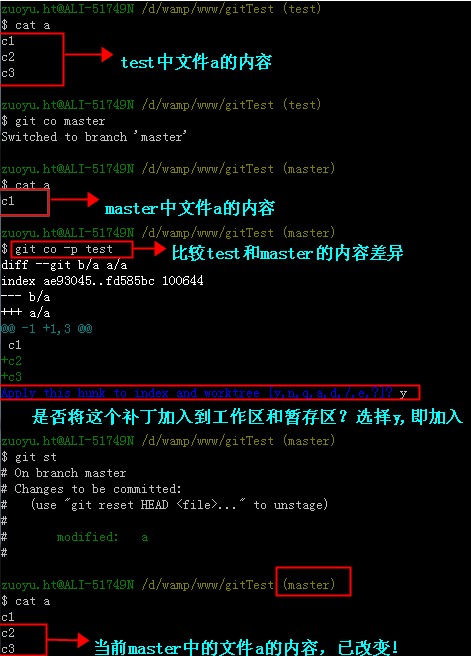
第一，如果当前分支和切换分支间的内容不同的话，容易造成冲突。

第二，切换到新分支后，当前分支修改过的内容就丢失了。

所以这个命令，慎用！

**6. git checkout -p <branch>**

这个命令可以用来打补丁。这个命令主要用来比较两个分支间的差异内容，并提供交互式的界面来选择进一步的操作。这个命令不仅可以比较两个分支间的差异，还可以比较单个文件的差异哦！



结束语：至此，关于git checkout命令暂告一段落，对于checkout命令，你也有所熟悉了吧。当然，git checkout还有一些其它用法，本文并没有讲到，你可以在git bash或终端中使用git checkout --help去进一步了解！

—————-

—————-

—————-

—————-

—————-